NASA-TM-108131

GODDARD
7N-61-TM
136177
P.7

X93-70912

EXPERIENCES IN THE IMPLEMENTATION OF A LARGE Ada PROJECT

Sally Godfrey
Code 552
Goddard Space Flight Center
Greenbelt, Md.20771
(301) 286-3600

Carolyn Brophy
Department of Computer Science
University of Maryland
College Park, Md. 20742
(301) 454-8711

BACKGROUND

During the past several years, the Software Engineering Laboratory (SEL) of Goddard Space Flight Center has been conducting an experiment in Ada [6],[8] to determine the cost effectiveness and feasibility of using Ada to develop flight dynamics software and to assess the effect of Ada on the flight dynamics environment. This experiment consists of near parallel developments of a dynamics simulator in both FORTRAN and Ada. A study team consisting of members from the SEL has monitored development progress and has collected data on both projects throughout their development.

Both the Ada and the FORTRAN teams began work in January, 1985, using the same set of requirements and specifications to develop their simulators. The FORTRAN dynamics simulator team completed acceptance testing by June, 1987, after following a development life cycle typical of projects in the flight dynamics environment [5]. The development was carried out on a DEC VAX-11/780 and the completed FORTRAN dynamics simulator consists of about 45,000 source lines of code.

The Ada development began with a period of training [7] in both the Ada language and the methodologies appropriate for Ada [11]. The team was not previously experienced in Ada, although they were more experienced than the FORTRAN team in both the number of years they had programmed (8.6 years compared to 4.8 for the FORTRAN team) and also in the number of languages they knew (7 compared to 3). The Ada team was also experienced in more types of software applications, but only 43% of the Ada team had previous dynamics simulator experience compared to 66% of the FORTRAN team.

Following the training period, the Ada team began a phase of analyzing the requirements and then they began design using an object oriented methodology called GOOD (General Object Oriented Design) which was developed by the team during the training and design phases. More information on GOOD and the lessons learned during the design phase can be found in [2], [4], and [10].

Coding and unit testing began in April, 1986, on a DEC VAX 8600 and continued through June 1987. The Ada project has completed system testing and consists of approximately 135,000 source lines of code[1]. This paper will describe some of the similarities and differences of the two projects and will discuss some of the interesting lessons learned during the code/unit test and integration phases of this project. END

INFORMATION COLLECTION

The information presented in this paper was collected by using the following four methods: 1) Collection of SEL forms 2) Interviews 3) Observation of development 4) Code analysis. The SEL forms solicit such information as a detailed breakdown of the hours spent by programmers, managers, and support staff on a project and detailed information on changes and errors which occurred during the development. During the course of the project, over 2000 forms were collected; about 625 of these documented errors and changes.

1. A source line of code is defined to be any 80 byte record of code including commentary, blank lines and executable code.

Each member of the Ada team (11 total) was interviewed individually to gain some insight into the experiences he or she had during implementation. Team members were asked questions concerning ease or difficulty of implementing features, unit testing, integration, correcting errors, using tools, etc. Questions concentrated on an individual's particular area of work, but general subjective questions were asked of the entire team. Observation of the development was accomplished by attending reviews and regular implementation meetings held by the team. These regular implementation meetings were actual working meetings in which team members discussed progress, solved implementation problems, clarified interfaces, shared knowledge, and planned implementation strategies. In addition, much information was gained through informal conversations with the team on implementation progress. Information received through code analysis was actually collected two ways. First, the code was examined to tabulate such attributes as number of modules, number of lines of code, number of comments, etc. Second, another Ada team, in the process of Ada training, performed code reading on parts of the dynamics simulator code as a training exercise and they provided their comments on the code.

The remainder of this paper will concentrate on some interesting comparisons between the FORTRAN and the Ada projects and some of the major lessons learned during the implementation phase of the Ada project.

1. FORTRAN/Ada PROJECT COMPARISONS

Several factors need to be considered when trying to directly compare metrics from the FORTRAN project and those from the Ada project. First, the FORTRAN project was considered to be the "real" operational version of the dynamics simulator being developed, and as such, it was necessary for that project to meet the schedules imposed by an impending launch date. The Ada team, on the other hand, was allowed a more relaxed schedule for development which included adequate training time, time to experiment with design methodologies, and finally, time to recode or enhance if "better" methods occurred to the developers. One result of this extra time was the development of a much more sophisticated user-interface for the Ada project.

Second, this general type of dynamics simulator was a very well-known application for the FORTRAN team since similar simulators have been built repeatedly in this environment. Thus, the general design of the FORTRAN simulator was reused from previous designs and was known to be a very satisfactory design for the application. In addition to the design, much of the code was reusable--about 36%. The Ada team developed a new design [1] which they felt was more suitable for Ada and which they felt more accurately represented the actual physical system they were trying to simulate. While this design may be a better physical representation of the problem, it did not have the advantage of previous use to refine and correct any possible problems. No Ada code was available for reuse but several FORTRAN routines were used by the Ada team. These comprised only about 2% of the code.

Keeping in mind these differences in the actual projects, we will discuss some interesting FORTRAN/Ada comparisons.

1.1 Size of Ada project is larger than FORTRAN project.

As mentioned in the background section, a simple count of the number of lines of code, including every line of any type as a line, yields a count of 135,000 source lines of code for the Ada project and a count of 45,500 source lines of code for the FORTRAN project. These figures are really a little misleading, since the Ada line count includes 23,000 lines of blank lines which are inserted for readability. Also, the Ada count includes 49,000 lines of comments compared to 19,500 lines of comments in the FORTRAN count. When the number of executable lines of code are compared, we find that the Ada project has 63,000 lines of executable code compared to 25,500 for the FORTRAN project.

In these particular projects, there were other reasons why the Ada project was larger. As we mentioned earlier, the Ada project was not constrained by schedule pressure and so they developed a system with more functionality--a system with more of the "nice to have, but not required" features. Naturally this increased the size of the system. To some extent, the Ada language itself was a driving factor for the size difference, since it requires more code to write such constructs as package specifications, declarations, etc. In addition, the Ada team used a style guide [3] that required certain constructs to be spread over several lines of code for readability.

Another interesting way to compare the size of the two projects is to examine the size of the load modules for each one. This also shows the Ada system to be larger-occupying 2300 512-byte blocks, compared to 953 512-byte blocks for the FORTRAN load module.

1.2 Project cost is similar for the two implementations.

One of the problems with trying to compute productivity is that there are many ways to compute it. Usually, in the Software Engineering Laboratory, the calculation is made by taking the total number of source lines of code developed and dividing by the number of hours spent on the project. The number of hours is carefully recorded on forms weekly and includes the hours spent on all phases of the project beginning with requirements analysis and ending with the completion of acceptance testing. In order to compare the FORTRAN and Ada projects, the calculations were made using the number of hours spent on each project from requirements analysis to the completion of system testing since acceptance testing has not yet been completed on the Ada system. As we see in figure 1, using the total number of source lines of code (SLOC) for each project, we get a productivity of 3.8 SLOC/hr. for the FORTRAN project and a productivity of 6.1 SLOC/hr. for the Ada project. Rememberin that the Ada code included many blank line of code that were not included in th FORTRAN line count, we recomputed the Ad figure, excluding the blank lines and got a productivity of 5.2 SLOC/hr. When we considered the effort required just t develop new lines of code and not th reusable code, the figures are 2.7 SLOC/hr. for FORTRAN and 6.1 SLOC/hr. for Ada with blanks and 5.0 SLOC/hr. without blanks. This would seem to imply that Ada is more productive, but we must remember that it took many more lines of code to develop the Ada system and that the style guide caused many Ada constructs to be spread over several lines.

Let's look at the figures when we consider only executable lines of code. Using only the number of lines of code which are executable, we got a productivity figure of 2.14 SLOC/hr. for the FORTRAN project and 2.8 SLOC/hr. for the Ada project. When we considered that many of the Ada constructs use more than one line, we looked at the number of executable statements (or semicolons) in the Ada project and recomputed productivity. Similarly for the FORTRAN, we counted statements and their continuations as one executable statement. Now we get a productivity of 1.85 SLOC/hr. for the FORTRAN project and .96 SLOC/hr. for the Ada project. Looking at the number of executable new statements in the FORTRAN yields a figure of 1.2 SLOC/hr. compared to .95 SLOC/hr. for the Ada project. These calculations would make FORTRAN look more productive.

FORTRAN

Lines of Code Used for Computation	Productivity
Total lines of code	3.8 SLOC/hr
Total lines of code excluding blanks	3.8 SLOC/hr
Executable lines of code	2.14 SLOC/hr
New lines of code	2.7 SLOC/hr
New lines of code excluding blanks	2.7 SLOC/hr
Executable statements	1.85 SLOC/hr
Executable "new" statements	1.2 SLOC/hr

Ada

Lines of Code Used for Computation	Productivity
Total lines of Code	6.17 SLOC/hr
Total lines of code excluding blanks	5.12 SLOC/hr
Executable lines of code	2.8 SLOC/hr
New lines of code	6.08 SLOC/hr
New lines of code excluding blanks	5.03 SLOC/hr
Executable statements	0.96 SLOC/hr
Executable "new" statements	0.95 SLOC/hr

Figure 1: Productivity Comparisons

Perhaps a better way of viewing the productivity problem is to examine it from the standpoint of cost to produce the product. The total cost of the FORTRAN project from requirements analysis through acceptance testing was about 8.5 man-years of effort. The Ada project cost, using actual figures from requirements analysis through system testing and estimating the acceptance testing cost, is around 12 man-years of effort. When we take into consideration the percentage of reused code in the FORTRAN project and assume all the code generated was new code, it would have taken about 11.5 man-years of effort to develop the FORTRAN system. This makes the cost of developing the two systems roughly the same, especially when we consider that the Ada project was a "first-time" project and that the Ada project had slightly more functionality than the FORTRAN.

1.3 Error types found in both projects show similar profiles.

Detailed information was kept on the types of errors found in both projects and based on 104 forms collected for the FORTRAN project and 174 forms collected for the Ada project, the error types show a similar profile. Figure 2 shows the distribution of error types for each project.

Error Type[a]	FORTRAN[b] %	Ada[c] %
Computational	12	9
Initialization	15	16
Data Value or Structure	24	28
Logic/Control Structure	16	19
Internal Interface	29	22
External Interface	4	6

[a]There may be more than one error reported on a form.
[b]104 forms
[c]174 forms

Figure 2: Error Profile

An example of a computational error might be an error in a mathematical expression. An error like using the wrong variable would have been classified as data value or structure error. Internal interface errors refer to errors in module to module communication, while external interface errors refer to errors in module to external communications.

Perhaps one result here that is suprising is that the team expected to have fewer internal interface errors with Ada, but the percentage is not significantly different from the FORTRAN. When the detailed information on the Ada errors was examined, we learned that many of the errors classified as internal interface errors were caused by a type change of some sort. For example, a variable may have been classified as one type in one portion of the code and a different type in another, or the original type chosen for a variable might not have been suitable. Another common reason that internal interfaces were changed was that a new function was added to the module which required an interface change. Also, in some cases, a developer would find he needed another variable from some other module which he did not originally think he needed.

1.4 The percentage of "very easy to find" errors was less in the Ada project than the FORTRAN project.

Detailed information was captured on the effort required to isolate errors .The error levels were categorized a) very easy or less than one hour b) easy or one hour to one day c) hard or one to three days d) very hard or more than three days. The FORTRAN team found that 81% of their errors were in the "very easy" to isolate category. In comparison, the Ada team found only 59% of their errors in that category. There are several possible explanations for this. First, many of the errors found by the FORTRAN team were types of errors which would have been identified by a more rigorous compiler such as the Ada compiler Throughout the project, the Ada team felt that the compiler was one of the most useful tools because it was able to pinpoint many errors at the early stage of compilation. Another possible explanation for the difference in effort to locate errors is the difference in experience of the teams with the language. The Ada team was not experienced in Ada and did not feel they had the same intuition as the FORTRAN team did to aid in isolating errors.

2. MAJOR LESSONS LEARNED DURING IMPLEMENTATION OF THE Ada PROJECT

2.1 A flat structure usually has more advantages than a nested structure. Thus, nesting should be used sparingly.

The object oriented design used by the team [9] seemed to promote a nested structure for information hiding purposes. While the nesting was not explicitly specified in the design, it seemed to be a natural manifestation of the object oriented design--so the parts of an object or a package would be included inside that package instead of being called in from the outside. The team felt that they were implementing nesting conservatively, and indeed, one view of the system shows that it has 124 packages of which 55 are library units. However, the nesting in the system was extensive--many levels deep in some places.

This amount of nesting caused many problems for the Ada developers. First, nesting increased the amount of recompilation necessary during implementation and testing. Many more units had to be recompiled when changes were made to the system since Ada assumes dependencies between nested objects or procedures even when there are none. Since compilation is a lengthy process, this slowed down the development process. Much unneccessary recompilation could have been avoided by the use of more library units.

Second, nesting increased the difficulty of unit testing. In fact, the greater the level of nesting, the more difficult the unit testing was. The lower level units were not in the scope of the test driver, and a debugger was necessary to "see" into these lower level units. For the purposes of unit testing in FORTRAN, a unit is defined as a subprogram. When this same definition was applied to the Ada, unit testing difficulties arose since many of these units could not be tested in isolation. Instead, it was necessary to integrate units which fit logically together, usually integrating up to the package level, before testing was done. Nesting also increased the difficulty of tracing problems since it is hard t identify the calling module of a neste unit.

2.2 A high degree of nesting was foun to be an impediment for reuse.

Perhaps the major advantage of usin library units instead of nested units i that their use increases the potential o reusability. When nesting is used, the siz of the compilation units, the componen sizes and the file sizes all tend to b larger. Thus when these larger units ar examined for potential reuse, it is muci more likely that only a portion of the larg unit will actually have the code whicl performs the needed function for the ne system. Then it becomes necessary to unnes the code before reuse is possible. Thi: unnesting is very labor intensive.

Another similar Ada project presentl under development in the SEL has examinec this project's code for reuse and has founc that it could use as much as 40% of the original code. However, it was necessary tc unnest all of this code before reuse. This use of library units would have enabled the second project to reuse the code directly.

2.3 "Call-through" units are not an efficient way to implement an object-oriented design.

"Call-throughs" are procedures whose only function is to call another routine. These were used to group appropriate modules exactly as they were represented in the design so that a physical module of code was created for every object in the design. Thus, when objects were nested inside objects, a "call-through" was used to get to the inner object. Implementation of "call-through" units could be accomplished using either nested or library units. This practice resulted in additional code which increased the system size and testing complexity. This unneccessary code could have been eliminated if some of the objects in the design were left as logical objects, rather than coding every object in the design to preserve the exact design structure.

2.4 An abstract data type analysis should be incorporated into the design process to control types.

Since the Ada team was not previously experienced in Ada, it took time to get accustomed to the strong typing of Ada. The tendency was to create too many types. A type would be created with a strict range for a particular portion of the application. Then other areas of the application would need a similar type, but the original one would be too restrictive. So another type was created, along with a corresponding set of operations. Some of the difficulty with this method of typing began to emerge during critical design,--where interface problems developed due to typing differences.

Multiple types also increased the difficulty of testing modules. Test drivers needed to be larger to handle multiple types and were often coded as large "case" statements in order to provide a testing capability for each type.

A recommendation for future Ada developments is to incorporate an abstract data type analysis into the design process to control the generation of types. A more general new type would be defined, then many subtypes of that type could be used in various sections of the application. This type analysis would provide the following advantages: 1) operations would be reused, 2) there would be fewer main types to manage, and 3) families of types would be developed that would inherit properties from each other.

SUMMARY

In spite of a lack of experience in Ada at the beginning of the project, the Ada team was able to develop a very suitable dynamics simulator in Ada which meets the requirements originally developed for the FORTRAN development effort. The overall cost of the projects appears to be similar and early indications of reuse potential in the Ada project are very encouraging. Most of the problems encountered by the Ada team are surmountable. Many are either caused by a lack of experience with Ada or an immaturity of the tools. Both of these problems will be resolved in time.

There are still many unanswered questions to be considered on this project--for example, nothing at all has been mentioned about maintainability, reliabilit or performance. It is still too early t look at these results on this project, but research efforts are continuing on thi project and several other Ada project in th SEL. Hopefully, these efforts will provid even more answers about the use Ada in th future.

REFERENCES

1. Agresti, W., Church, V., Card, D., et al. "Designing with Ada for Satellit Simulation: A Case Study," *Proceeding of 1st Annual Symposium on Ad Applications for NASA Space Station* Houston, Texas, June 1986.

2. Brophy, C. and Godfrey, S.,et. al "Lessons Learned in the Implementatio Phase of a Large Ada Project, *Proceedings of the Washington Ad Technical Conference*, March 1988.

3. Goddard Space Flight Center Ada User' Group. *Ada Style Guide (Version 1.1)* Goddard Space Flight Center document SEL-87-002, June 1987.
4. Godfrey, S., and Brophy, C. *Assessin the Ada Design Process and It Implications: A Case Study*, Goddar Space Flight Center document, SEL-87-004, July 1987.

5. McGarry, F., Page, G., et. al. *Recommended Approach to Softwar Development*, Goddard Space Fligh Center document, SEL-81-205, April 1983.

6. McGarry, F., and Nelson, R. *A Experiment with Ada-The GRO Dynamics Simulator*, Goddard Space Flight Center, April 1985.

7. Murphy, R. and Stark, M. *Ada Training Evaluation and Recommendations*, Goddard Spoace Flight Center, October 1985.

8. Nelson, R. "NASA Ada Experiment--Attitude Dynamics Simulator," *Proceedings of Washington Ad Symposium*, March, 1986.

9. Seidewitz, E. and Stark, M. "Towards a General Object Oriented Software Development Methodology," *Proceedings of 1st International Conference on Ada Applications for the Space Station*, June 1986.

10. Seidewitz, E. and Stark, M. *General Object Oriented Software Development*, Goddard Space Flight Center document, SEL-86-002, August 1986.

11. Stark, M. and Seidewitz, E. "Towards a General Object Oriented Ada Lifecycle," *Proceedings of Joint Conference on Ada Tech/Washington Ada Symposium*, March 1986.